FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Combining Fuzz Testing and Spectrum-based Fault Localization

Carlos Daniel Coelho Ferreira Gomes

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática

Supervisor: Professor Rui Filipe Lima Maranhão de Abreu

March 20, 2022

Combining Fuzz Testing and Spectrum-based Fault Localization

Carlos Daniel Coelho Ferreira Gomes

Mestrado em Engenharia Informática

March 20, 2022

Resumo

O software tem crescido a nível de complexidade. Cada vez mais existem diferentes bibliotecas e programas que se vão encaixando uns nos outros. Consequentemente, o software apresenta mais e mais linhas de código e, cada linha de código pode causar algum tipo de erro, sendo mais suscetível a acontecerem erros durante a execução dos mesmos. Por isso mesmo, os desenvolvedores têm que criar testes manualmente para detetar a existência de algum tipo de erros e, dessa forma, evitar elevados custos futuros.

Fuzzing, geração automática de dados de entrada, é uma técnica que permite os desenvolvedores encontrar vulnerabilidades que muitas vezes não são encontradas com apenas os testes que são regularmente feitos pelos programadores.

Quando são encontrando erros no programa ou comportamentos estranhos do mesmo, os programadores têm que fazer perceber, encontrar e corrigir o código correspondente a essa falha. Para automatizar essa tarefa, técnicas de localização de falhas automáticas foram criadas, sendo as técnicas de localização de falhas baseadas no espectro do programa as mais comuns.

Assim, com este trabalho, juntamos estas duas técnicas que bons resultados têm demonstrado separadamente, e que até agora, nunca se tentou perceber se as mesmas funcinariam em sincronia.

Para atingir este objetivo, juntamos um programa de cada campo. Jazzer foi a ferramenta de Fuzzing escolhida para exercitar os novos testes através de dados de entrada inválidos, e Gzoltar como uma ferramenta de localização de falhas em software através do espectro do programa. Para ligar ambas as ferramentas, desenvove-mos um pequeno programa que traduz os ficheiros gerados pelo Jazzer para um formato que o Gzoltar fosse capaz de utilizar. De seguida, realizamos um estudo sobre o sistema de forma a compreender o quão valioso e útil este poderia ser, percebendo se esta abordagem pioneira devolveria uma excelente área de estudo que ajudaria o dia a dia dos programadores.

Keywords: Testes, Fuzzing, Localização de falhas de Software

Abstract

The software has grown in complexity. More and more different libraries and programs are used to create other ones. Consequently, the software presents more and more lines of code, and each line of code can cause some error, being more susceptible to errors during their execution. For this reason, developers have to create tests manually to detect the existence of some errors and, in this way, avoid high future costs.

Fuzzing, the automatic generation of input data, is a technique that allows developers to find vulnerabilities that are often not found with just the tests that programmers regularly do.

When errors are found in the program or strange behaviour is detected, programmers have to understand, find and correct the code corresponding to that flaw. Automatic troubleshooting techniques have been created to automate this task, with program spectrum-based fault localization techniques being the most common and presenting good results.

Thus, with this work, we put together these two techniques that have shown promising results separately, and that until now, no one has ever tried to understand if they would work in sync.

We have put together a program from each field to achieve this goal. Jazzer was the Fuzzing tool to exercise the new tests through invalid input data. Gzoltar was the spectrum-based fault localization software tool. In order to connect both, we developed a small program parsing the Jazzer output to a format that Gzoltar could use. Moreover, we performed a study about the system itself to perceive how valuable it could be, understanding if this new path demonstrates an excellent area of research that may help the programmers' daily lives.

Keywords: Testing, Fuzzing, Software Fault Localization, Spectrum-Based Fault Localization

Acknowledgements

This Dissertation was a massive challenge because it was a bit different from the usual. Usually, the document is started in course *Dissertation Planning* that precedes *Dissertation*. In my case, everything was developed in one semester in the course *Dissertation*. Therefore, I have to thank my supervisor, Professor Rui Maranhão de Abreu, for all patience and guidance throughout this work.

My parents and sister, I have no words to describe their unconditional support, emotional and financial, especially during the last two years.

Next, I want to use this opportunity to exhibit my gratitude to some groups of people who allowed me to reach this end of a chapter and were there for me when I most needed it. To "Os Lindos", Gonçalo, Teresa, Diogo and Inês, thanks for all the conversations, for all study hours, for all parties, jokes and drinks. To "Simple fellows", António, Pedro, Tiago, Álvaro and João, I know you are good people, most of the time. I want to thank you for all ignored messages, for all wake-up calls, for the fellowship and for that sofa in Rio Tinto. To "The Council", Joana, Joana and Eduardo, thanks for the daily motivation, the carry during the Masters and for making me addicted to term.ooo . Next, to "The Road" thanks for all the cleared doubts, all the trash talk, wasted hours in senseless activities and all the friendship. Finally, from my hometown, Beatriz e Sara, thank you for all the support.

Finally, I am really grateful to this incredible faculty for allowing me to live these incredible five years and for all people who cross my life and watched me grow.

Carlos Daniel Gomes

“Just because you can’t prove it, doesn’t mean it’s not true.”

Robert Graysmith

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

API	Application Programming Interface
SBFL	Spectrum-Based Fault Localization
SFL	Software Fault Localization

Chapter 1

Introduction

Contents

1.1 Context

Software testing is the primary source of finding defects and failures and ensuring that the software matches all the features it is supposed to, with no unexpected behaviours. Although, software engineers face growing challenges in building and maintaining increasingly complex systems. For any typical development project, roughly 60% of software costs are development costs, and the remaining 40% are testing costs. For custom software, testing costs often exceed development costs [38]. However, even with all the effort in testing, software bugs may appear during the development phase, or in the worst-case scenario, in the customer's hands, causing lots of damage.

Testing, debugging, and verification are intense tasks during the development phase. Widely used to accomplish those tasks are unit tests. They verify each small part of a program. The developer has to understand all the faults raised by them, what is causing them and how to correct them. Nowadays, there are different domains where localization has been applied [3, 4] and several tools and frameworks are helping on test creation and debugging like JUnit [1], Hamcrest [2], JaCoCo [3], and the IDEs themselves. These tools provide a set of manual assist to find errors and their location in the code. Nevertheless, in complex systems, manual findings become infeasible. Several Software Fault Localization techniques automate this process, finding the code block most

[1]https://junit.org/
[2]http://hamcrest.org/
[3]https://github.com/jacoco/jacoco

likely to be faulty [40]. A tool created in the Faculty of Engineering of the University of Porto (FEUP) is Gzoltar [8]. Gzoltar is a toolset for JAVA that implements the Spectrum-Based Fault Localization technique. It can track the executed code and identify the blocks that do not behave normally, providing information about the error localization, using all the passing and failing unit tests created in the project.

As mentioned before, the test's development seizes the major of the time of a project. The automatic generation of invalid inputs to test software known as fuzz testing was proposed in 1988 [28], which can help to create several types of tests. Nowadays, fuzzing is an automatic testing technique that covers numerous boundary cases using invalid data (from files, network protocols, application programming interface (API) calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities [25]. With fuzzing, a software engineer can discover profound software errors not exposed by manual testing. Yet, the engineer has to perceive what is causing the error.

1.2 Motivation

Software testing and debugging consume too much time in the lifetime development cycle. This bug hunting is crucial to delivering a reliable product to the customer. Therefore, it is critical to decreasing the chase time for an attested program.

Spectrum-Based Fault Localization and Fuzz Testing are two proven techniques in the fight to reduce the time consumed in testing and debugging. Although, there is no way to connect both processes. Suppose a developer could generate many valid and invalid inputs, detect when the program behaves incorrectly, and get the why and where that behaviour happened. In that case, the time for testing and debugging will instantaneously decrease.

1.3 Objectives

The main objective of this works is the design and integration of fuzz testing and spectrum-based fault localization techniques. This tool, created for Java programs, must accept and/or generate all the inputs necessary to feed the Gzoltar toolset, tests that might be already in the target project and return all the feedback from the SFL technique. The program must be as automatized as possible.

Besides the tool creation, the integration of both techniques may not retrieve all goods from both worlds. It is essential to assess the quality of the produced diagnostic reports before the integration happens and after that. Both techniques may be excellent, but together may not.

In the end, the tool will be available for the daily testing and debugging phase of the development cycle at a business level as well as for the academic and research level.

1.4 Document Structure

The document presents five chapters where the Introduction is the first. Excluding this one, the rest of the document is as follows:

- Chapter 2 describes state of the art in the topics about Fuzzing Test and Spectrum-Based Fault Location techniques and tools
- Chapter 3 details the proposed solution and consequent implementation, discussing all the choices and limitations.
- Chapter 4 presents all the results and the process behind their generation.
- Chapter 5 follows a final reflection about the work done, pointing out the work contributions and future tasks.

Chapter 2

State of the Art

Contents

2.1 Introduction

In order to create a complete state of the art, we defined some keywords to guide the search for articles that might present the intended content. Thus, we used the following statements:

- Software Fault Localization
- Spectrum-Based Fault Location
- Fuzzing
- Java Fuzzing
- Fuzz Testing

- Gzoltar

Farther, Google Scholar[1] and Association for Computing Machinery Digital Library[2] (ACM DL) were the two search engines mainly used to discover new items. Therefore, we applied the mentioned sentences to the search engines obtaining the basis results. Following, we adopt a crossing of the snowball approach - findings through the article references chapter - and the citation method - findings through the list of articles that reference the article. The snowball approach was predominantly used in recent articles, whereas with older ones citation method was applied. The majorioty of the articles came out from the ACM Digital Library[3], Springer Link[4] and IEEExplore[5].

2.2 Fuzzing

At the University of Wisconsin-Madison in the 80's decade, Miller et al. created a program that generates a random characters stream and saves it into a standard output file [28]. The authors call to this program *fuzz*. Then, they could use the file generated as input of a program and check the program behaviour. This was the first concept introduced regarding the generation of the random test.

Since its first application, the *fuzz* research field has improved a lot with different types of fuzzers for every language and paradigm. According to Liang et al., this area has polynomial growth of interest over the years, showing an increased number of publications regarding fuzzing and fuzz testing. This made fuzzing and fuzzers evolve to different states but similar concepts. Boehme et al. say that fuzzing is an automatic bug and vulnerability discovery technique that continuously generates inputs and reports those that crash the program [5]. Notwithstanding, Godefroid refers to fuzzing as an automatic test generation and execution to find security vulnerabilities[13]. Li et al., says that the fuzzing test is the generation of massive normal and abnormal inputs targeting applications and trying to detect exceptions by running them on target applications and monitoring the execution states [23]. These definitions, although identical, have some differences. The first two only mention the generation of invalid input, and the last one underlines the creation of valid and invalid input. To avoid discrepancies and unify all the concepts, Manes et al. created definitions based on several works from reputable conferences [26]. They have the following definitions:

- Fuzzing - Execution of the program under test using inputs sampled from an input space that protrudes the expected input space of the software being tested.

- Fuzz input - Input that the program under test may process incorrectly and trigger a not intended behaviour.

[1]scholar.google.com
[2]dl.acm.org
[3]https://dl.acm.org/
[4]https://link.springer.com/
[5]https://ieeexplore.ieee.org/

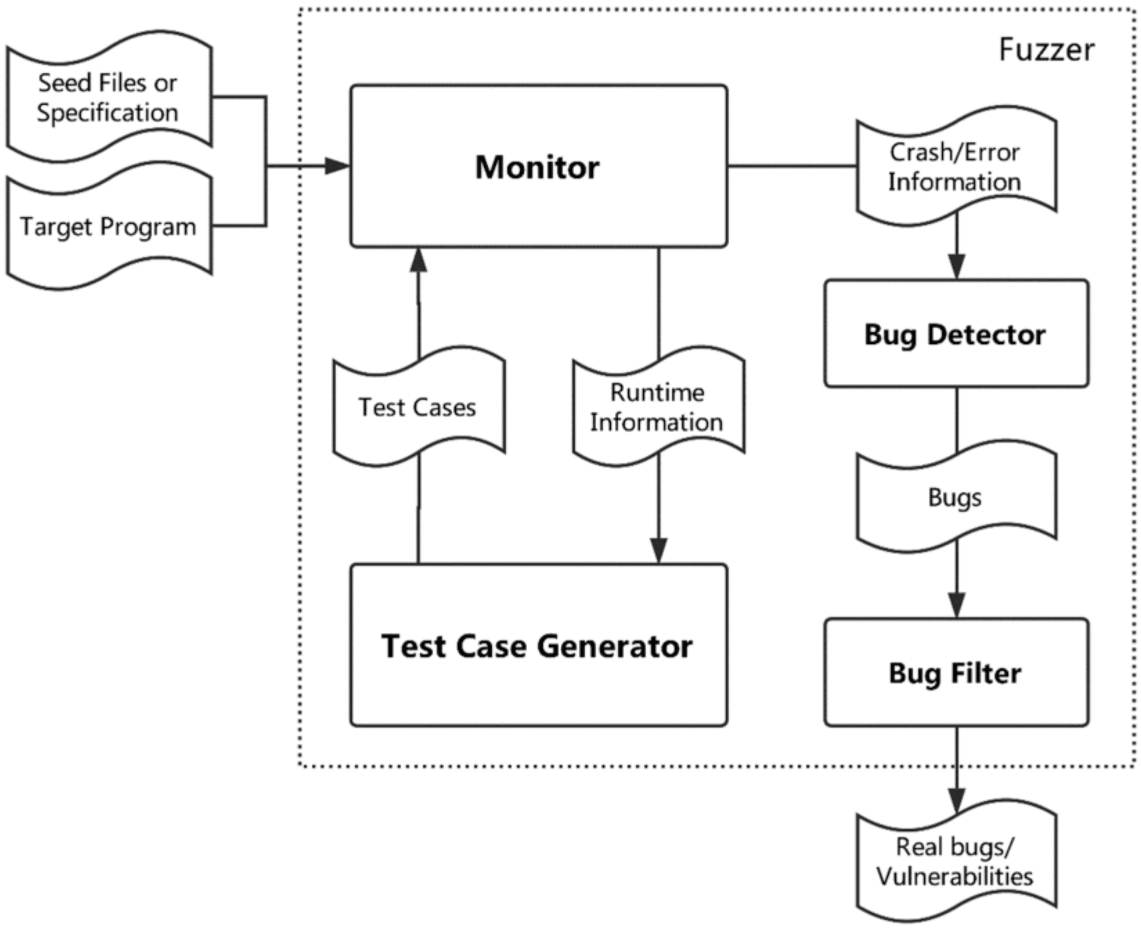


Figure 2.1: General fuzzing process. Figure source [25]

- Fuzz Testing - Use fuzzing to verify if a program violates its correctness policy.
- Fuzzer - Tool that runs a fuzz testing technique.
- Fuzz campaign - Executing a fuzzer on a program with a specific correctness policy.
- Bug Oracle - Program that verifies if the execution of a system violates a correctness policy.
- Fuzz Configuration - Parameters that control the fuzz algorithm execution.

From now on, we will keep using these definitions in this chapter whenever possible.

In the beginning, fuzzers generate random inputs with no criteria and without any knowledge from the target. Today this is entirely different. Several fuzzers can collect in-depth information about the target or just partial information. This level of enlightenment a fuzzer has about the target divides the fuzzers into three huge categories: White-Box Fuzzer, Grey-Box Fuzzer and Black-Box Fuzzer.

Figure 2.1 illustrates the generic fuzzing process. The monitor receives the initial input or an input specification and the target, run the seed and gathers all the run-time information, such as code coverage and taint data flow. The information is then used to generate more tests through mutational-based or grammar-based methods and then sent to the monitor. When a crash or an error occurs during the Fuzz Campaign, it is analyzed to understand if it is a failure and a new bug is detected.

2.2.1 Black-Box Fuzzing

Black-box Fuzzing was the first technique being implemented and created the first concept of Fuzzing. This concept lies on generate inputs blindly and without any information about the program under test. Besides, black-box fuzzers always needs a start input, known as seed input, that they will change afterwards. Several articles [13, 23, 25, 26] mention the low effectiveness of Black-Box fuzzing in finding deeper and complex bugs in the code. This type of fuzzing can be very quick finding shallow bugs even in complex system, it is easy to implement and execute, without any major pre-requirement to start.

Even not knowing anything about the program under test, Black-box fuzzers may use two different ways to generate the tests: mutational generation or grammar-based generation. Mutational generation picks the seed and randomly mutate them in randomly places creating new inputs to the program. Example of mutations are bit flips, bit removals or bit copies. Grammar-based Fuzzers[13] are also known as Model-Based Fuzzers[26] and Generational Fuzzers[5]. This type of Fuzzers use a model that describes how the generated input must be. This way, it is possible to fuzz an application that accepts a specific and/or complex data structures. For example, applications that parses XML documents. Black-Box Fuzzers with grammar-based generator shows better performance, effectiveness and accuracy in bug finding than the totally blind approach.

2.2.2 White-Box Fuzzing

Black-Box random fuzzing is practically limited, and grammar-based fuzzing is labor intensive [13]. In addition, with Black-Box fuzzing the user is never capable to tell when is secure to say that a program does not need more fuzzing. There is no way to formally prove that every possible path of a system was executed, i.e., that is not possible to prove mathematically that is no more bugs in the code. To overcome this issue, in 2008 Godefroid et al. presented for the first time a White-Box Fuzzing, a new approach using Dynamic Symbolic Execution also called Concolic Execution [26]. It starts with well-formed inputs, performing a symbolic execution concurrently with a dynamic execution [7, 14]. These executions gather constraints on inputs from conditional branches. Then, the constraints are systematically solved with constraints solvers whose solutions map new program executions path. Next, this process is repeated using search techniques to try to go through all possible paths of a program. This approach is capable of analyse all the internal characteristics of a program exploring every single possible program state. This line of action gather some challenges that is mentioned in the Subsection 2.2.5.

2.2.3 Grey-Box Fuzzing

The last main classification for Fuzzing is Grey-Box Fuzzing. As the name suggests, these fuzzer types mix white and black Fuzzing. Instead of running blindly like the Black-Box approach or with full knowledge like White-Box solutions, Grey-box Fuzzers utilizes lightweight static analysis techniques to get some internal information about the program under test. This way is possible to save some computational resources and retain some information. The principal techniques used in Grey-Box Fuzzing are Code Instrumentation and Taint Analysis. With instrumentation, the fuzzer can gather code coverage of the system in runtime, and with taint analyses, it can trace the taint data flow. These techniques will allow different details to be analyzed, contributing to better mutating or generating inputs to improve the fuzzing campaign results.

2.2.4 Other Classifications

There are other classifications regarding other steps in the fuzzing process. In Subsection 2.2.1 we already mentioned two different classifications regarding the input generation, Mutational Fuzzing and Grammar-based Fuzzing.

Li et al. refers to the definition of Code-Coverage Fuzzing and Directed Fuzzing. Code-Coverage Fuzzing has one objective: to make the generated inputs return the maximum code coverage value. It assumes that more bugs are probably found by running more and deeper source code. Directed Fuzzing refers to fuzz to reach a particular target inside the code. They can be defined at the beginning of the fuzz or during the target execution [23].

Godefroid introduced the Hybrid Fuzzing classification. This classification is given to those fuzzers who try to get the best of White-Box Fuzzing and Grey-Box Fuzzing. Hybrid Fuzzers try to find execution points where more straightforward techniques are more reliable than heavy ones. On the contrary, the fuzzer must use more complex techniques where the simpler can not give adequate information [13].

Another possible classification Feedback-Based Fuzzing. This nomenclature is widely used to define fuzzers like Jazzer or LibFuzzer. Feedback-Based Fuzzing gathers any Fuzzer that consume or analyze the program and its behaviour in runtime or through code pre-processing. All Code-Coverage Based Fuzzer are Feedback-Based Fuzzers. One particular case is kAFL [36]. kAFL uses feedback from Intel's Processor Trace (PT) technology to guide the fuzzing process.

Besides all these concepts, the Fuzzing tends to improve not in a macro perspective, i.e., the new fuzzers are using as much info as they can get from the binary code or source code without unbalancing the performance. How this information is gotten and used to improve bug discovery is the key to improving Fuzzing from any perspective.

2.2.5 Challenges

The Fuzzer research field growth has been focused on the improvement of the techniques already implemented or the introduction of new methods of code analyses which will improve the generation of the tests and consequently the finding of vulnerabilities.

Although, every step on a Fuzzer creation has its challenges. The key challenges identified are the following:

1. How to get initial input
2. How to update or generate new input
3. How to select the best seed from the input pool
4. How to detect a bug
5. How to filter the bugs found

Regarding the initial seeds, some fuzzers do not even need them. Usually, Grammar-based Fuzzers do not need seeds, but they do need a specification file about the input, like QuickFuzz [17], and Dharma [10]. Mutational-Based Fuzzers need an initial seed input. Although Fuzzers like AFL or AFL++ are not Grammar-Based Fuzzers but are capable of starting the fuzzing process without any corpora. Herrera et al. discuss that in complex real-world problems, a good initial corpus is essential. It does not mean that more bugs will be found, but the initial fuzzing process will be faster because non-useful inputs were discarded. Massive corpora with no selection criteria may also lead to skipping shallow bugs. So, state-of-the-art fuzzers have minimization techniques - reducing the seeders size without changing the execution behaviour - allowing discarding unnecessary inputs with some pre-defined heuristics like the execution time and code coverage reached.

As mentioned by Herrera et al., Mutational-Based Grey-Box Fuzzers are the most widely used technique in this research field [20]. There are very different types of mutations that can be applied to the seed. AFL was an innovator in this area. It applies several different types of mutations like bit flips, arithmetic mutation, re-combinations with varying lengths and stepovers, sequential addition and subtraction of small integers and sequential insertion of known interesting integers like 0, 1, INT_MAX [23]. If not controlled inside a seed, these mutations will lead to a random mutation in structured files, which may lead them to fail the validation inside the program under test. This validation problem is not an issue in Grammar-based fuzzer because they know the inputs structures. However, mutational ones may not be capable of dealing efficiently with file checksum, magic bytes, version number checks, or other possible validations [23]. Taint analyses and deep neural networks may be used to predict the changes in the internal input location.

Understanding which seed will be first used with a big input pool is important. The good choice will provoke an effective fuzzing process leading to a faster vulnerabilities exploitation or to a better exploration returning more useful and accurate information. AFL assign to each seed an energy value. These values will differ among all the fuzzers. AFL prioritizes small seeds that show higher code coverage and branch coverage. Böhme et al. extend this AFL approach with Markov Chains [6]. It initially prioritizes new paths explorations and then the exploitation . Other techniques like taint analyses and static analyses that try to discover faulty parts of the code can be used. They are usually used in directed fuzzing for obvious reasons. These techniques are correlated to the input generation and can not be dissociated from them.

Different seeds may lead to different program behaviours, expected or unexpected ones. Thus, program languages leave certain unwanted behaviours being treated like usual runs instead of crashes. So, beyond the typical crashes inside the programming language, there were created sanitizers. Sanitizers are programs that detect unsafe or undesirable behaviours and treat them as violations. For example, Jazzer [9] uses AddressSanitizer [18] and Undefined Behavior Sanitizer [11].

Creating vast inputs will lead to similar ones or the same bug through different program paths. This problem is called *deduplication*. According to Manes et al., there are three major answers: stack backtrace hashing, coverage-based deduplication and semantics aware deduplication. Stack backtrace hashing technique saves the backtrace of the crash and assigns it a hash value. This is the most widely used and oldest technique. It may not save all the stack backtrace, and it can select specific frames of the stack or even use all the information connected to it. Coverage-based deduplication relies on the heuristics used by the fuzzer in the code coverage, which may not be accurate to bug triage. Semantics-aware Deduplication uses the data-flow analyses of each crash to mark different violations [26].

Besides the technicalities in the fuzzer tool construction, Boehme et al. addresses issues in [5]. Fuzzing has been a field of research without many specific measures to validate. As mentioned before, mutational fuzzing is being studied empirically. There is a lack of specific measures to stop fuzzing or classifying a fuzzing campaign has finished ensuring the program under test has no more vulnerabilities or that the most important bugs were found. So, even with all advances in fuzzing, we have no way to prove that software is secure.

2.3 Software Fault Localization

Software is more and more complex, with more lines of code and more likely to have failures. A failure in a program is an event that makes the program not run as expected. The responsible for these events are software failures, also known as bugs or faults. When these events occur, the developers have to find them and fix them. Usually, this bug hunt is done through techniques like program logging, assertions, breakpoints and profiling [40]. Program Logging is still a very used technique; it saves the information into files or prints it in the monitor when abnormal behaviour is detected. Assertions are program statements used during development that must be true. Otherwise, they will match with unwanted behaviour. Breakpoints allow the programmer to stop the program during execution, changing the program's state and observing each step of execution. Profiling is related to debugging unexpected memory issues and performance.

These traditional techniques are still used, but they are too tedious and consumes too much time of the developing cycle. Over the years, there were created and researched several techniques in order to detect the root cause of software faults. Spectrum-based Fault Location (SBFL) is one of the most researched techniques in the last decades. According to Wong et al., 35% of the articles analysed in "A Survey on Software Fault Localization" corresponds to SBFL researches.

Mnemonic	Name	Description
BHS	Branch Hit Spectra	conditional branches that were executed
BCS	Branch Count Spectra	number of times each conditional branch was executed
CPS	Complete Path Spectra	complete path that was executed
PHS	Path Hit Spectra	complete path that was executed
PCS	Path Count Spectra	number of times each definition-use pair was executed
DHS	Data-dependence Hit Spectra	definition-use pairs that were executed
DCS	Data-dependence Count Spectra	number of times each definition-use pair was executed
OPS	Output Spectra	output that was produced
ETS	Execution Trace Spectra	execution trace that was produced

Table 2.1: Different types of program spectra [19]

2.3.1 Spectrum-Based Fault Localization

Spectrum-Based Fault Localization is a technique influenced by probabilistic and statistical models. As the name implies, the techniques use the program spectra, a collection of data that provides a specific view on the dynamic behaviour of software [24]. It gathers information targeting a specif test suite [16]. Therefore, information like statements, branches, paths and basic blocks are kept and then some statics are calculated, such as binary coverage status, how many times a statement was covered and many others, specifying for each piece of information if it was collected from a failing or a passing test [34, 33]. Moreover, all this different code-coverage information will specify a different type of spectra [19]. Figure 2.1 shows do the different types of spectra identified by Harrold et al.. Although every type of information can be generalized as block hit spectra [2]. Thus, for a single run, all the information aforementioned can be maintained in a block hit spectra with a size equal to the number of blocks calculated, where every element is a flag representing if that part of the code was covered for that specific execution. So, every single program execution can be kept in a spectra matrix where each line is an execution and each column is a part of the code. Then, each execution is gathered in a one-column matrix with the information if the execution failed or did not. Figure 2.2 represent the binary matrix for a generic program with M parts and N executions.

With all the necessary info gathered, the *suspicious values* can be calculated. Suspicious Value is a number associated with each block system analyzed on the spectrum, which quantifies

$$\begin{array}{ccc} & M\ Components & \begin{array}{c} error \\ detection \end{array} \\ N\ spectra & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

Figure 2.2: Activity Matrix containing all information about block hits and execution error [2]

the possibility of the fault being in that block. To calculate the suspicious value, a good number of formulas are proposed. Some articles mentions four as the best metrics [40, 41, 32, 30]: *Ochiai* (equation 2.1)[1], $D*$ (equation 2.2) [39], OP^2 (equation 2.3) [29] and *Tarantula* (equation 2.4) [21].

$$Ochiai = \frac{e_f}{\sqrt{(e_f + e_p)(e_f + n_f)}} \tag{2.1}$$

$$D^* = \frac{e_f^2}{n_f + e_p} \tag{2.2}$$

$$OP^2 = e_f - \frac{e_p}{e_p + n_p + 1} \tag{2.3}$$

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}} \tag{2.4}$$

The equations above correspond to popular techniques and are using the notation used by Ghosh and Singh as follows:

- e_f block executed in a failing execution
- e_p block executed in a successful execution
- n_f block not executed in a failing execution
- n_p block not executed in a successful execution

After calculating every suspicious value, the technique can compute a ranking ordered from the highest to the lowest. Then, the developer goes through the ranking, checking block by block, looking for the failure, and in a good result, the bug may be in the first lines decreasing the effort needed to debug.

Meanwhile, there is no consensus about which metric is the best for all cases. It is unanimous that any formula is the best. One may have a better performance than the other in a context but be outperformed by the same technique in another one [32, 27]. However, we did not find any article that specified under what circumstances each formula would be the most indicated. Nonetheless, Pearson mentions in its evaluation of fault localization techniques article [31] that the choice of spectrum-based technique does not matter. His results demonstrated that all techniques evaluated perform almost identically with small insignificant differences.

2.3.2 Spectrum-Based Fault Localization Tools

There is some tools available Tarantula was one of the first tools applied in real projects using the SFL technique with the same name. Although, Tarantula is a tool for C programs.

Regarding Java programs, a well-known tool is Gzoltar [8]. It is available as an Eclipse plug-in, an Visual Studio Code Extension and a command-line interface tool. This tool retrieves a ranking with the suspicious values per line of code using the Ochiai formula. In addition, it provides a graphic interface that helps the developer debug the software.

In 2018, Ribeiro et al. presented *Jaguar*[35]. Aiming Java programs, this tool differs from other tools in spectrum collection. Jaguar gathers control-follow and data-flow spectrum information without increasing the run-time overhead significantly. Although, the definition-use associations (duas) connected to data-flow control can not be used in a whole program, leaving out of the data spectrum local duas and non-handled exceptions that are thrown during execution. In addition, Jaguar uses ten different metrics to calculate the suspicious values in duas and lines of code.

The latest SBFL tool, as far as we know, is Flacoco [37]. This tool is similar to Gzoltar. Both use the Ochiai formula to create the suspicious line ranking. Although, Flacoco offers an easy way to introduce different suspicious metrics formulas. Besides the design decisions, these tools differ mainly in the way they make the information available and the way they instrument the programs: Gzoltar has its instrumentation, and Flacoco relies on JaCoCo[6] framework. Gzoltar offers a graphic data visualization and three different interfaces, while Flacoco is available only as a command-line interface and a Java API.

2.4 Summary

In Section 2.2 we address an overview about the Fuzz Process, the current technologies regarding Fuzz Testing in Java Programs and some challenges in this field. Next, Section 2.3 approaches the Spectrum-Based Fault Localization, noting some of the current techniques and tools.

[6]https://www.jacoco.org/jacoco/

Chapter 3

Problem and Proposed Solution

Contents

This chapter aims to delineate the raised problem, how we address its issues, and the proposed solution implemented.

In order to expose the problem and solution, we split up this chapter into three sections. First, section 3.1 exposes what led us to develop this work and its outcome. Following, section 3.2 addresses the proposed and developed system to contest the problem, going deep in the environment established to the research, the small program that assists and connects the two independent modules, and the system pipeline. Lastly, we finish with a summary of this chapter in the section 3.3.

3.1 Problem and Goals

Software testing and debugging occupy much time in the software development life cycle. In order to ease this task, some valuable tools try to decrease the time and effort spent in this phase. Fuzz testing helps the engineers find unpredictable errors and behaviours in the programs, generating a huge set of inputs to test the software. Software Fault Location is a useful and informative technique that reports which block of code are most likely to be faulty. Then, engineers are capable of generating a significant amount of tests, and if they go wrong, they are also capable of understanding why and where that happens. However, there is nothing between these two steps, and this absence is the problem that guides this dissertation. So, this work aims to allow and connect two technologies, Fuzzing and SFL.

The two valuable tools presented are powerful separated, but they can not be so good together. It is important to understand if they will cooperate without denying their good performances if creating this connection. Besides, it is essential to perceive if the information created through this process will be helpful for the test developers, helping all the software development.

3.2 System

The proposed solution follows the chart in the Figure 3.1. After the creation of the fuzz driver, the user can run the Jazzer fuzzer. After that, the generated tests and the fuzz driver are passed to the tests generator tool, that will create a test suite based on Junit framework. Following, the Gzoltar will run the SFL technique with the generated tests outputting the sfl report with all the information needed to help the user in the debugging. With the report, the user can see where the error lies with a specific probability.

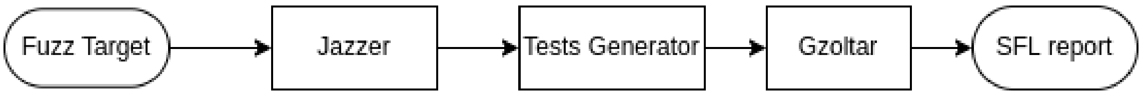


Figure 3.1: System Flowchart

This pipeline structure allows keeping each module independent. This independence grants us to cover different sets of situations besides our main goal of connecting both technologies. Software development testing is not only performed by a single person. It is known that a team usually carries out this task, so the different moments of the pipeline process might be performed in different timelines or even attached to different teammates. We are aware that this is not the focus of this study, but we can consider different perspectives. So, if a user wants only to get some inputs, it can use only the fuzzer. If there are already many unit tests created, the user can just run them. If the user already has the fuzzer outputs, the process can start in the tests generator phase, and so on. This flexibility will be important to compare results between different metrics when testing our solution, allowing compare different variables even during real-world software development.

3.2.1 Fuzzers

Regarding the fuzzers choice, we wanted to have as many fuzzers as possible. As evident, the fuzzer has to be a fuzzer capable of fuzzing Java targets because the well-known fuzzers like AFL or AFL++ are good, but they need the correct sanitisers to fuzz a java program. A Java program runs inside a Java Virtual Machine, so if we try to run the program, what is going to be fuzzed is the JVM itself. Following, the fuzzer has to retrieve random java primitives and not only bytes. As we want to fuzz classes during the development software phase, if the only input is bytes, the user would have to spend more time converting this type of data into the primitives he needs. Then, it would be nice for the fuzzer to accept custom generators, i.e., programs that retrieve structured data like an XML string or its representation in a specific class. Finally, through the previous research in the chapter 2 the fuzzer must be code-coverage based because they present better results in the errors finding. With all the constraints, we came up with three fuzzers to make part of this tool: Jazzer, JQF and Kelinci.

Kelinci generates file inputs, so it just retrieves arrays of bytes. Initially, we wanted to have an AFL-based fuzzer like Kelinci, but its performance in early experimentations was inadequate, rounding the 9.6 executions per second. Another metric that made us leave Kelinci as an increment and not an essential feature was that it is no longer supported. The repository has had no changes in the last four years.

JQF allows different input generators. It has capable of associating different *guidances*. This property would be perfect to our work. However, it has a considerable disadvantage. The only way to run the fuzzer and reproduce a crash is through its Maven plugin. There was no way to use the different generated inputs because JQF saves the input in its own way. The suggestion given by the creator of this fuzzer was to save the input in the fuzz driver as a readable human string. Although, this would be mandatory for classes to have a toString method and parser to backwards this action. So, if we are not capable of get the inputs to the test files we could not use this fuzzer.

Finally, Jazzer has checked all our constraints. It eases the input insertion in the new test files, and it has got good results. This fuzzer is still under development, so it only checks crashes in the program. Although it gets updated every week, and the developers pay attention to community feedback. One drawback to using this fuzzer is that it is new and open to bugs that may confuse the user of our system and the system itself. Besides, it is mandatory to use Jazzer API to get the input to the driver, but it has very useful methods with different primitives to feed the program under test, and it is the same API that let the user to retrieve the specific input generated for a certain execution.

3.2.2 Tests Generation

The Tests Generation is a program written in java whose function is only to generate tests in JUnit to feed the Gzoltar component.

This module is the only one that did not exist before this work. This component opens a bridge between the SFL Module and the Fuzzing Module, translating, creating and adapting the fuzz drivers to the testing framework utilized by Gzoltar. As the communication media between the fuzzers and the SFL tool, this module receives as input the driver and the bad results got from the fuzzer.

We are using JavaParser to get the driver code and create a new java file, the test file, with the dependencies associated with the driver and the dependencies needed to the test run without problems, or in this case, to fail without problems. So, all the imports declared in the driver are replicated in the test file.

The program reads the driver and figures out the driver's file typology. Initially, the whole system was planned to run three different fuzzers, where each one has a different syntax, so following the driver's content, we could understand what we had to add to the test suite. However, we found some limitations on how the inputs were generated and kept by each fuzzer and how could we get the actual input. This is explored in the subsection 3.2.1. Following, we set a test function for each input generated, which matches the block code written in the original file. Next, in the first line of the block statement, we add the input declared in the same variable as the fuzz driver to do not conflict with any other variable. This method allows us to create each test quickly and without flaws, without considering problems as flow control in loop statements, where the driver can consume input multiple times. Finally, the file is saved in the directory specified by the user or in the default one.

This simple and effective generation method permits generating the tests quickly, not occupying time that can be spent on more important matters like the first process or the last one.

3.2.2.1 Architecture

This subsection aims to explain in detail how exactly the JUnit tests are generated. The figure 3.2 presents the important classes created to achieve this goal.

- TestFile - Class that represents the new file that will be generated. TestFile holds all the information about every piece of code to insert in the new file. After getting all the inputs and information needed, the method save() it's called to terminate save the file and terminate the program.

- TestSource - Class representing the fuzz driver. It reads all the information regarding the fuzz driver file and passes it to the TestFile class. This handles find by itself what fuzzer driver is it reading.

- SourceType - Enumeration regarding the different types of fuzz drivers the program can receive

- Input - Abstract class that handles the reading of all the inputs. Each fuzz driver has its own subclass because the class has to find only the correct files to read.

- BlockType - Class responsible to build the lines of code corresponding to the actual input in the tests. Different inputs need different processing so there is one subclass for each type of input, i.e., for each fuzz driver type.

- TestMethod - Class holding information regarding the methods of the test class created. So, for each input read is created a new instance of this class matching with the method in the test class.

It is important to mention that there are three more classes that do not are mentioned in the list above because they are not an active asset in the test cases generation. Although, they are useful

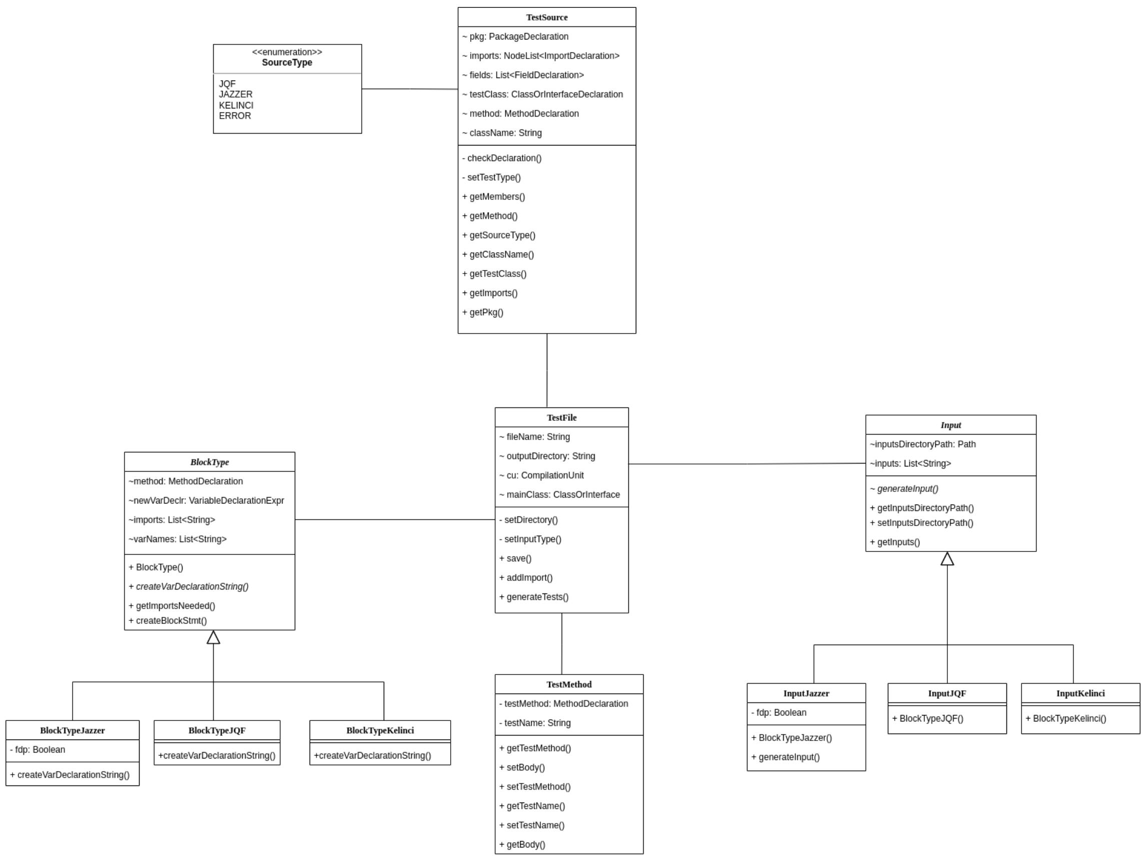


Figure 3.2: Class diagram

for parse the parameters passed to the program, to hold the JavaParser configuration and to be the program entry point.

3.2.2.2 Inputs

As mentioned in section 3.2, the Tests Generation gets as input the output of the fuzzer module. But, the input tests are not the unique parameters that this component accepts. The program usage is as follows:

"Usage: –inputDir=<> –outputDir=<> –fuzzFilePath=<> [–className=<>]"

There are three mandatory arguments to start the program:

- –inputDir - the path where the input created by the fuzzer is kept.
- –outputDir - the path where the user wants to save the test class file generated by the program.
- –fuzzFilePath - the path for the fuzz driver.

The only optional argument is *–className*. This argument if provided will match to the class name of the class generated. If not, the class name will be inferred from the path given in the argument *–fuzzFilePath*.

3.2.2.3 Outputs

Tests written with the JUnit framework are the output of this component. Every method added to the test class has the pattern *test(index)*, where *index* is given by the program and matches with the order that the software reads the inputs directory starting at 0. The class name follows a pattern as well, *(Fuzzer Driver Class Name)Test*. In Figure 3.3, we can see the fuzzer only found one bad input for the driver *JsouFuzzer*, and it is represented in the method *test0*. For multiple flawed input findings, the class will have one method for each one of them with the pattern mentioned.

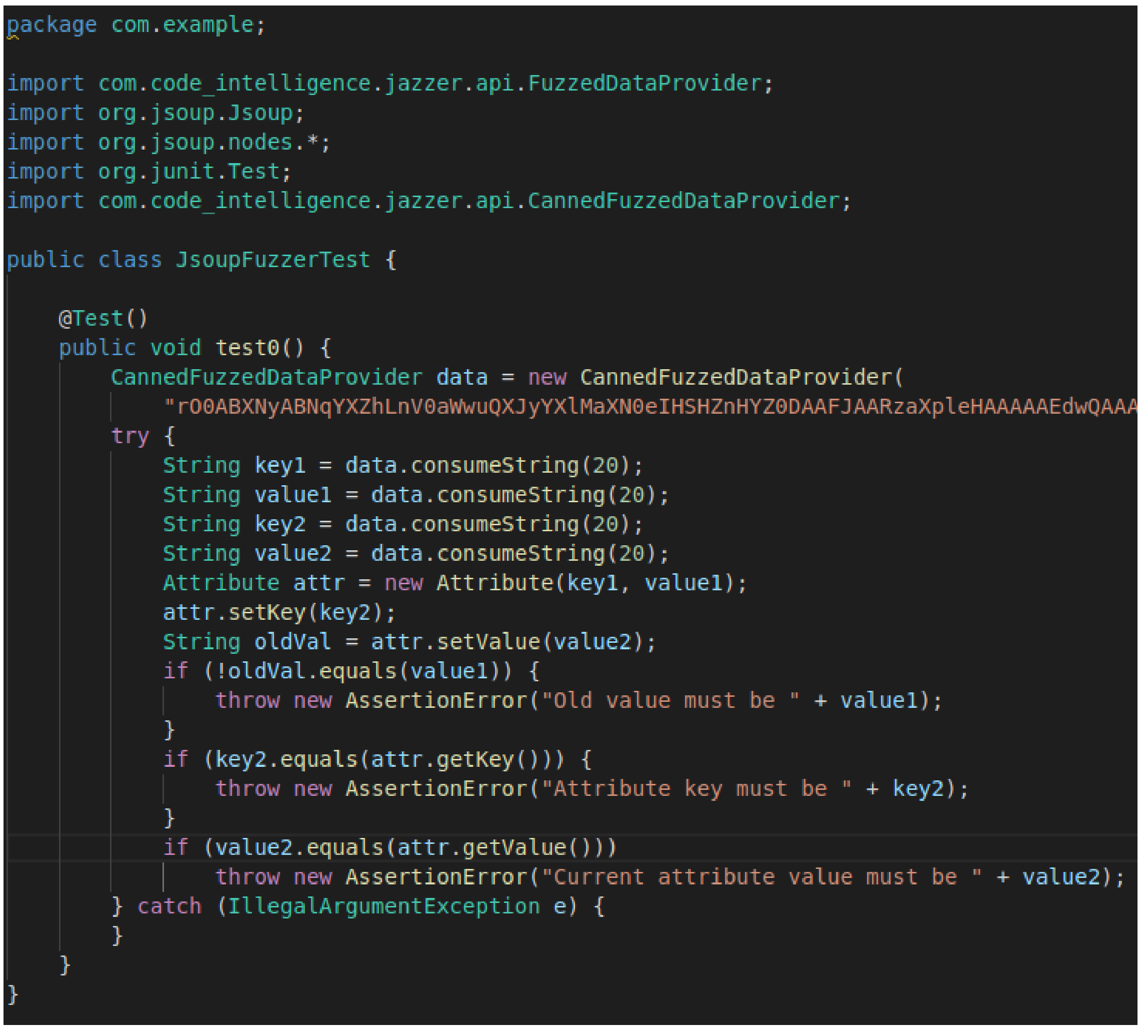
The actual input generated, as mentioned before, is written in each method passed to the new test file. The string that can not be completely seen in Figure 3.3 is the base 64 string that initiates the input. The variable associated with this string is the variable *data* that has the same name as the argument in the fuzzer driver method. This variable is an instance of the class *CannedFuzzed-DataProvider* whose behaviour is the same that the one created inside the fuzz driver, providing the test with the exact same instances.

Besides the information seen, everything in the fuzzer driver is passed to the output. So, if the fuzz class target has any fields or methods, all is copied to the new test class keeping the content in full, to avoid some undeclared variables, functions or fields.

3.2.3 Gzoltar

Regarding the existing tools to execute Spectrum-Based Fault Localization, we identified three: Gzoltar, Jaguar and Flacoco. All of them showed good results in finding bug locations. Jaguar is very different from the other two. It can perform ten different metrics to generate the suspicious values, which could be used in our study. Although this tool is only focused on being used as an Eclipse plugin, we needed a tool that could be used standalone. In addition, we did not find any more articles where this tool had been used before; there was a lack of confirmation in its effectiveness.

Flacoco and Gzoltar are two similar open-source tools. Both of them apply the Ochiai formula to compute the suspicious ranking. The significant differences between these two tools were how they instrumented the program under test and how they were available to the community. Flacoco takes advantage of a Java code coverage library, JaCoCo, using their code coverage heuristics, while Gzoltar uses its own instrumentation methodology. Gzoltar is available as a Maven plugin, Ant task, Visual Studio Code Extension, Command-line interface. Flacoco is provided as a Java API and as a Command Line Interface. What led us to use Gzoltar versus Flacoco was the lack of articles supporting Flacoco. Gzoltar was the most reliable option supported by the use in several articles.



```java
package com.example;

import com.code_intelligence.jazzer.api.FuzzedDataProvider;
import org.jsoup.Jsoup;
import org.jsoup.nodes.*;
import org.junit.Test;
import com.code_intelligence.jazzer.api.CannedFuzzedDataProvider;

public class JsoupFuzzerTest {

    @Test()
    public void test0() {
        CannedFuzzedDataProvider data = new CannedFuzzedDataProvider(
            "rO0ABXNyABNqYXZhLnV0aWwuQXJyYXlMaXN0eIHSHZnHYZ0DAAFJAARzaXpleHAAAAAEdwQAAA
        try {
            String key1 = data.consumeString(20);
            String value1 = data.consumeString(20);
            String key2 = data.consumeString(20);
            String value2 = data.consumeString(20);
            Attribute attr = new Attribute(key1, value1);
            attr.setKey(key2);
            String oldVal = attr.setValue(value2);
            if (!oldVal.equals(value1)) {
                throw new AssertionError("Old value must be " + value1);
            }
            if (key2.equals(attr.getKey())) {
                throw new AssertionError("Attribute key must be " + key2);
            }
            if (value2.equals(attr.getValue()))
                throw new AssertionError("Current attribute value must be " + value2);
        } catch (IllegalArgumentException e) {
        }
    }
}
```

Figure 3.3: Example of a generated test

The Gzoltar command-line interface was the interface used under this experiment. It provides different output files, and the one used was the one whose function was to save the ranking generated by the application.

3.3 Summary

This chapter discusses the problem, a possible approach to solve this issue, and the implementation followed.

In Section 3.1, we explain the lack between two good technologies that ease the testing and debugging in real-world problems. To face this problem, we proposed a solution described in Section 3.2. The three modules associated with our implementation are detailed in the following subsections, with a focus on Subsection 3.2.2 that corresponds to the component created by us to connect the other two.

The solution described will be evaluated in the next chapter to understand if we have reached our goal.

Chapter 4

Results and Discussion

Contents

This chapter aims to present the results of the tests performed to achieve the goals described in the previous chapter and test the effectiveness of the tool developed. In Section 4.1 we address the metric used to evaluate the system and how the tests were performed. In Section 4.2, all the outcome is analyzed and discussed if it is favourable for the work goals. Following, Section 4.3 approaches the threats that may appear due to some testing decision. The chapter ends with Section 4.4 where we do a summary of the chapter.

4.1 Methodology

In order to understand the effectiveness and accuracy of the system created at 3, we decided to use the Defects4J[1] [22]. This bug database allows us to reproduce a bug behaviour and see the code change that corrected the bug. This framework will help us have a basis for reproducible bugs that can be found and located in the code. So, our testing procedure is performed as the following:

- Select a project version to fuzz

[1]https://github.com/rjust/defects4j

- Create the fuzz driver to find the specific bug
- Run the system
- Inspect the sfl report generated and check if the code changed to eliminate the bug matches the rank generated.

As mentioned in Section 3.2, the tests passed to Gzoltar can be unit tests already created by the project developers or unit tests generated by our system. Those tests can be failing or passing because a project may have failing tests unrelated to the bug belonging to that specific version. In order to understand the impact of the tests generated by our system in the failure discovery and localization we used two different input sets:

1. Developers unit tests, passing and failing
2. Not failing developer unit tests and tests generated by our system

Item number 1. allow us to have a comparable basis, i. e., how the SFL technique answer without the fuzzing phase, obtaining a basis regarding the use of Gzoltar as an independent tool. In number 2, we only use those generated by our system for the failing tests to understand the system's effectiveness with minimum outside interference. It is important to mention the need of those tests that do not fail because the Fuzzer is not capable of save the valid input.

So, to Fuzz the Defects4j project versions we created a fuzz driver needed to Jazzer being capable of starting the fuzzing campaign. This driver is inspired in the project failed tests cases that triggers the bug. This way we will be capable of finding the bugs through fuzzing and check the locations predicted for the bug. The bug localization is constructed from the file lines difference between the bugged project and the fixed one, using a set of scripts already available in a gitlab repository[2].

4.1.1 Environment

All the experiments discussed in this chapter were carried out in a Lenovo YOGA 530-14IKB with an instance of Ubuntu 20.04.3 LTS 64-bit, with a Intel® Core™ i7-8550U CPU @ 1.80GHz × 8. The different system components are running under Java 11. For the tests, JUnit 4.12 is used. The Jazzer and Gzoltar versions match the version available at the time of use of each tool Github repository. Both tools are used as jars with all the dependencies needed to work standalone. Gzoltar commit version is e2b581c516bf02d406f9fef2e51bdb612dfcd77f and Jazzer commit version is 0a80fb8af1726d1e30b12cec7500af82941f10d9.

Regarding the system options, we decided to cap Jazzer running time and its maximum JVM memory occupation. Then, each fuzz driver runs for ten minutes, or until the JVM reach the 8 gigabits of memory. Gzoltar only uses the online experimentation. These configurations let us to keep every test under the same constraints.

[2]https://bitbucket.org/rjust/fault-localization-data/src/master/

4.2 Results

4.2.1 Tests found

As comprehensible, the tests that feed Gzoltar will impact the outcome. The impact in the system in an overall perspective can be in the quantity and the quality of a testing suite. To better understand the results, we refer to the quantity of tests the actual number of tests that each test suite has. The quality of the tests is connected to the number of tests that are actually connected to the bug target. So, a good test suite may have many tests that find the bug target, improving the Gzoltar statistics.

If the project has only one bug detected by the failing tests, the code lines performed most likely will be connected to that bug. On the one hand, the more tests Gzoltar has on the same bug, the more accurate and precise it will be in the SFL report because the lines that might have the bug are performed more than the others. On the other hand, if the failing tests find multiple bugs in the same suite, the SFL report decreases its efficiency in detecting precisely a specific bug, because all the flaws not related to the bug target are being considered. So a test is as good as possible if it has a good number of tests per bug.

In Table 4.1, are identified the number of bugs that every project has. The objective of defects4j, as explained before, is to have a database with projects controlled versions where each version has a specific real bug. The column "Unique bugs on the project" confirm the specified bug's existence. The column "Bugs found with fuzzing" tells us about the bugs found in the project through the first step of our system. We can see that at least the bug target was found using fuzzing in every project. Although, other bugs not specified in the database were found and may not even be related to the bug target. This issue is addressed in the next subsection.

When we say that all bugs are found, the table may not agree with the statement. There is only one project where Jazzer is not capable of finding the bug, *Jsoup_91b*. This is because the test that fails in the developer test suite is not even running the actual code. This bug is related to the Jsoup library trying to parse all the files that it receives, so when it tries to parse binary files, for example, it throws a Null Pointer Exception (NPE). The test and the code correction to this bug is a workaround: it simply checks if the file is binary and does not touch the content. So, the fuzzing in this target finds the problem, and the SFL tool tries to answer it.

Another circumstance regarding the tests and the SFL report is the number of tests used to find one bug. As mentioned in Chapter 2, the number of failing and passing tests is important will have a direct impact on the report. In Table 4.2, we can see the number of failing tests and errors raised per project. Each developer failing test suite (oc) has one or two failing tests, while the failing suite found by Jazzer (op) often has multiples. These observations will be important in the SFL report. Although, gathering this information together allows us to say that the first module works good like expected and that its transition to Junit is done successfully. However, we can know what we expect from the tests in this controlled environment and what we do not. In a real case, we only know the expected behaviour of a system and expected unexpected behaviour could only be created synthetically, not replicating the system's actual condition.

Project	Unique bugs on project	Bugs found with fuzzing	Different Bugs Found
Codec_11b	1	1	0
Codec_13b	1	1	0
Codec_16b	1	1	0
Codec_18b	1	1	0
Csv_15b	1	1	0
Gson_11b	1	1	0
Gson_12b	1	1	0
Gson_15b	1	2	1
JacksonCore_16b	1	2	1
JacksonCore_17b	1	1	1
JacksonCore_19b	1	1	0
JacksonCore_8b	1	1	0
JacksonCore_9b	1	2	1
Jsoup_79b	1	4	3
Jsoup_80b	1	1	0
Jsoup_81b	1	2	1
Jsoup_84b	1	2	1
Jsoup_86b	1	1	0
Jsoup_89b	1	1	0
Jsoup_91b	1	1	1
Math_101b	1	1	0
Math_93b	1	1	0
Math_94b	1	1	0
Math_96b	1	1	0
Math_98b	1	1	0

Table 4.1: Different Bugs found in each project

Project	Tests	Error	Quantity
Codec_11b	oc	org.apache.commons.codec.DecoderException	2
	op	org.apache.commons.codec.DecoderException	2
Codec_13b	oc	java.lang.NullPointerException	2
	op	java.lang.NullPointerException	1
Codec_16b	oc	java.lang.IllegalArgumentException	1
	op	java.lang.IllegalArgumentException	1
Codec_18b	oc	java.lang.StringIndexOutOfBoundsException	2
	op	java.lang.StringIndexOutOfBoundsException	3
Csv_15b	oc	org.junit.ComparisonFailure	1
	op	java.lang.AssertionError	1
Gson_11b	oc	com.google.gson.JsonSyntaxException	1
	op	com.google.gson.JsonSyntaxException	1
Gson_12b	oc	java.lang.ArrayIndexOutOfBoundsException	2
	op	java.lang.ArrayIndexOutOfBoundsException	1
Gson_15b	oc	java.lang.IllegalArgumentException	1
	op	java.lang.AssertionError	1
		java.lang.IllegalArgumentException	1
JacksonCore_16b	oc	junit.framework.AssertionFailedError	1
	op	java.lang.ArrayIndexOutOfBoundsException	3
		java.lang.AssertionError	2
JacksonCore_17b	oc	com.fasterxml.jackson.core.JsonGenerationException	1
	op	java.lang.StringIndexOutOfBoundsException	1
JacksonCore_19b	oc	java.lang.ArrayIndexOutOfBoundsException	1
	op	java.lang.ArrayIndexOutOfBoundsException	2
JacksonCore_8b	oc	java.lang.NullPointerException	1
	op	java.lang.NullPointerException	1
JacksonCore_9b	oc	junit.framework.ComparisonFailure	2
	op	java.lang.ArrayIndexOutOfBoundsException	11
		java.lang.AssertionError	3
Jsoup_79b	oc	java.lang.UnsupportedOperationException	1
	op	java.lang.IndexOutOfBoundsException	1
		java.lang.NullPointerException	14
		java.lang.StringIndexOutOfBoundsException	11
		java.lang.UnsupportedOperationException	1
Jsoup_80b	oc	java.lang.IndexOutOfBoundsException	1
	op	java.lang.IndexOutOfBoundsException	170
Jsoup_81b	oc	org.junit.ComparisonFailure	1

Continued on next page

Table 4.2: Number of test failled tests per project

Project	Tests	Error	Quantity
	op	java.lang.AssertionError	1
		java.lang.NullPointerException	15
Jsoup_84b	oc	org.w3c.dom.DOMException	1
	op	java.lang.NullPointerException	11
		org.w3c.dom.DOMException	3
Jsoup_86b	oc	java.lang.IndexOutOfBoundsException	1
	op	java.lang.IndexOutOfBoundsException	146
Jsoup_89b	oc	java.lang.NullPointerException	1
	op	java.lang.NullPointerException	1
Jsoup_91b	oc	java.lang.AssertionError	1
	op	java.lang.NullPointerException	5
Math_101b	oc	java.lang.StringIndexOutOfBoundsException	2
	op	java.lang.StringIndexOutOfBoundsException	1
Math_93b	oc	junit.framework.AssertionFailedError	1
	op	java.lang.AssertionError	1
Math_94b	oc	junit.framework.AssertionFailedError	1
	op	java.lang.AssertionError	1
Math_96b	oc	junit.framework.AssertionFailedError	1
	op	java.lang.AssertionError	1
Math_98b	oc	java.lang.ArrayIndexOutOfBoundsException	2
	op	java.lang.ArrayIndexOutOfBoundsException	2

Table 4.2: Number of test failled tests per project

4.2.2 SFL Reports

Regarding the SFL reports, as mentioned in Section 4.1, we compute two different reports. The important data to consider in these reports is the ranking of the first lines. So, we will analyze the first ten lines of each report. These are the lines at the top of the file and might be the first lines that a developer might use to debug the program. Figure 4.1 compares the number of target lines that there are in the first ten lines. The target lines are those there are added to fix the program bug. We noted that the bug correction is an addition of code most of the time during the experiments. So, many times the line target is in the report has the following line, or the previous one. We decided to add a margin of error to the supposed correction to fill in this issue. The line targets are so those that differ between versions with two lines of error from and to that line. For example, if the original line is set to line 200, the lines considered in our calculation are the lines in the interval [198-202].

So, looking at Figure 4.1 we can see that the percentage of target lines that are in the top ten of each report is not much higher in the most of the cases. Although, those that are at the top are in a very good position which means that they have a relative good suspicious value (see Figures 4.3 and 4.2). Even with a good set of tests, we have to focus on two key aspects in integrating fuzzing test and SFL: Multiple Bug Finding and Quantity of Tests found. We manually checked every report and test generated to confirm the data generated.

The Multiple Bug Finding is connected to the number of bugs found by the Fuzzer module in the fuzzing phase of the system. The Tables 4.1 and 4.2 scrutinize the number of bugs found and each bug in each project. There are projects where the number of bugs found is superior to the number of bugs expected, which is one due to the test environment. The finding of multiple bugs is a good achievement to the system, but it confuses the SFL tool when it tries to localize the bug target, spreading all the suspicious values through all the lines performed, which may not even be correlated if the bugs are not related to each other. Our major case is the project *Jsoup_79*. As expected, this project has lines at the top regarding only the developer tests with a high number of Suspicious Value, Figures 4.3 and 4.2. However, when we switch to the tests generated, we see that the project has much more bugs than they are not related. The paths exercised will be different from the bug target.

The Quantity of Tests generated matches with the number of tests created for a specific bug. So regarding the percentage of target lines in the top ten, we noticed that for the same bug, if the number of generated tests increases, the report will be directed to those bugged lines. For example, the Fuzzing process founds in the project *Jsoup_86b* the bug specified in defects4j database. Furthermore, it finds 146 different tests raising the suspicious value of the lines almost to 1.0. However, as mentioned in Chapter 2, the fuzzers have a bug triage system whose job is to understand if the bug was already found, and for that specific code coverage, there are no more tests generated. We perceived this action in the project *Codec_13b*, where the developers suite has two similar tests allowing the suspicious values of the lines to be a little bit higher, but the rank remains the same. Despite the small increase of tests in the developers' suite, the suspicious value differences are not much different from that found by the fuzzing campaign suite.

Looking at the figures provided, we find two projects where the target lines do not show in the top thirty. As explained before, the developers' tests in the *Jsoup_91* do not run the actual code, and the correction is a workaround. More important to analyze is the project *Codec_16b*. This project correction is different. This correction is a change in a constant that is not covered by the tool. This line does not appear in the SFL report. Although, the top rankings in this project are similar, showing a good certainty in the piece of code that throws the error.

Finally, the project *Jsoup_84b*. This project do not show any target line in the top ten. In addition, the number of tests and bugs found are different from the developers' suite. Although, it is important to mention that the developers' suite SFL report retrieves a bad result where the target line is in top twenty-two, but with the same suspicious value of each line in the top forty.

Looking at the data, we can say that the system works as expected from an overall perspective. Even with a bit of initial effort to setup, joining these two different modules does not show a downgrade in the bug finding, showing equal or better results with the manual tests.

4.3 Limitations

Some variables may influence the results. As mentioned before, the fuzzers do not provide data that do not behave as expected. They only save those tests that crash the program. Although, the SFL tool can only work correctly if provided with passing and failing tests. In our experiences, we used the passing tests from the developers' test suite in both cases. Nevertheless, in real-life projects, there are no tests initially. Even this system provides a way to find faster the bug localization, it assumes and needs a set of passing tests.

Regarding the fuzzing process, Jazzer needs a driver to know which methods to call with the data generated. Every driver in this experimentation was created manually and inspired by the test that triggered the developers' suite error. Even with this information, every driver took much time to create because a driver has to be adapted to the library that he belongs, and we have different libraries with different functions. In addition, the driver specification influences the fuzzers outcome, and consequently, the tests passed to the SFL tool.

4.4 Summary

In this Chapter, we analyze the tests experimented with exposing the proposed system's viability. In Section 4.1, we start explaining how do we procedure every test. Following, we cover a discussion and presentation of some statistics whose task relies on showing if the system is a feasible solution. In Section 4.3 we address what can restrict the usage of the system in a real context, and what can derail the results demonstrated.

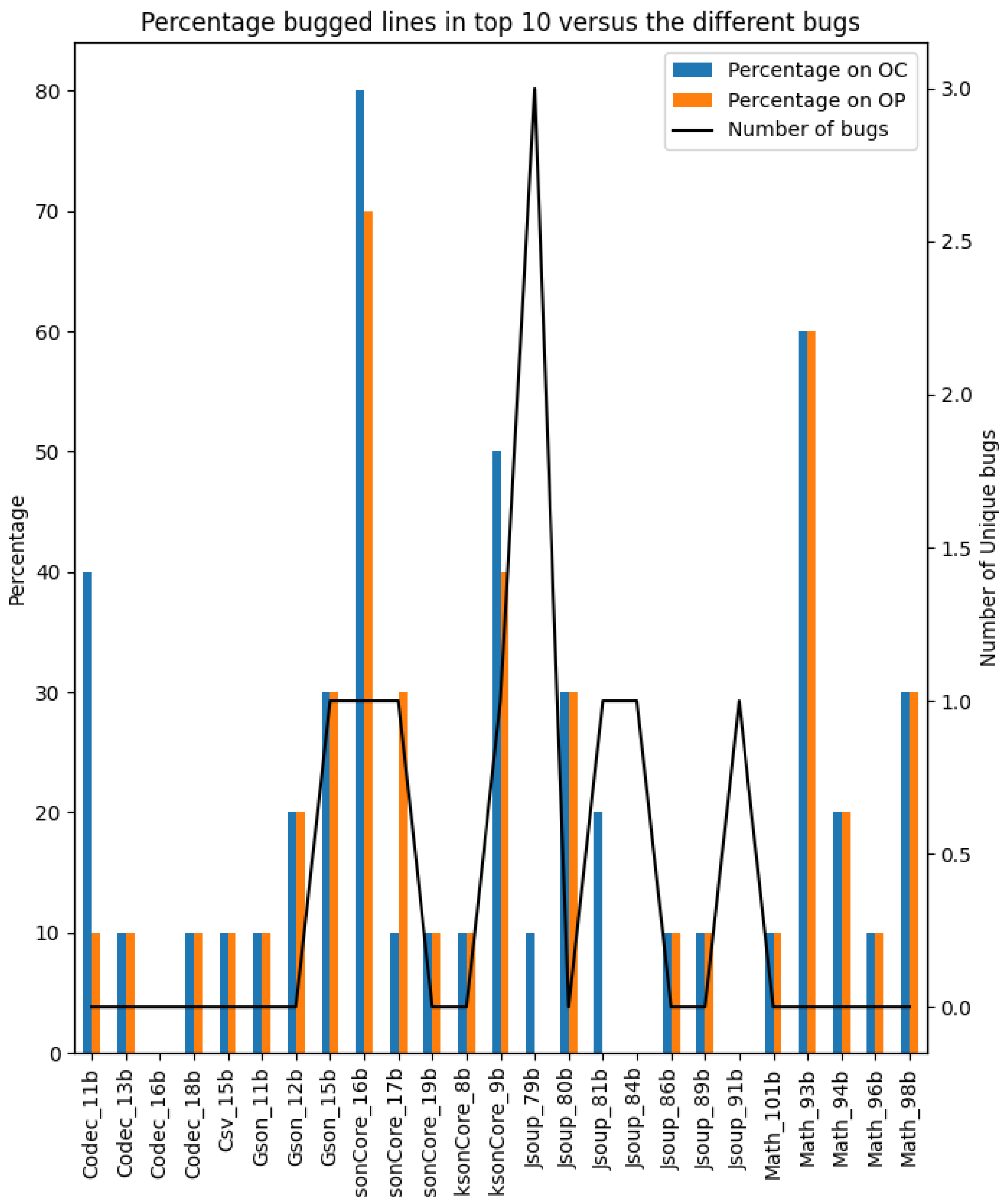


Figure 4.1: Percentage of lines in the first 10 lines of the SFL report

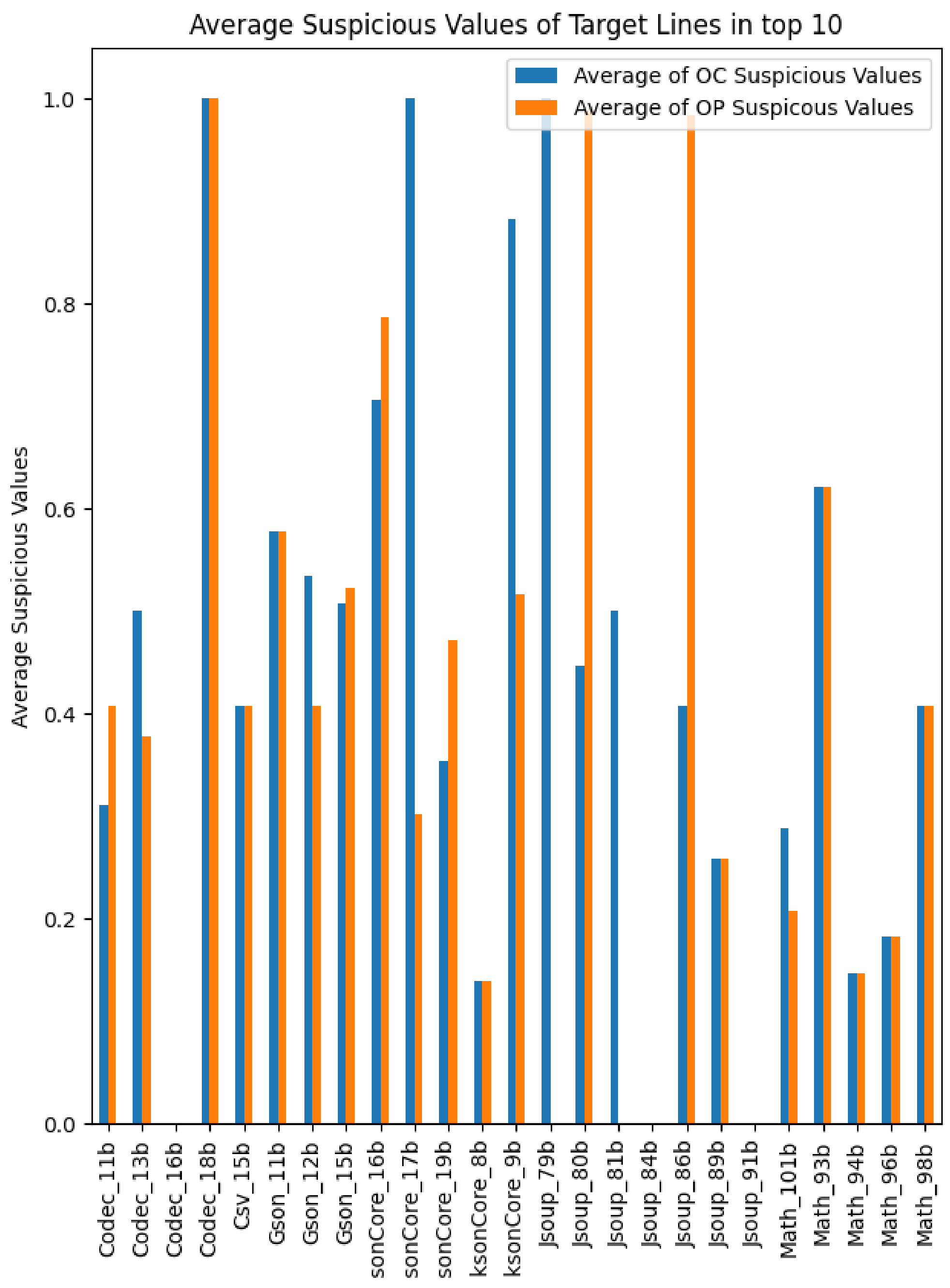


Figure 4.2: Average Suspicious Values of the Target Lines located in the first 10 lines of the SFL reports

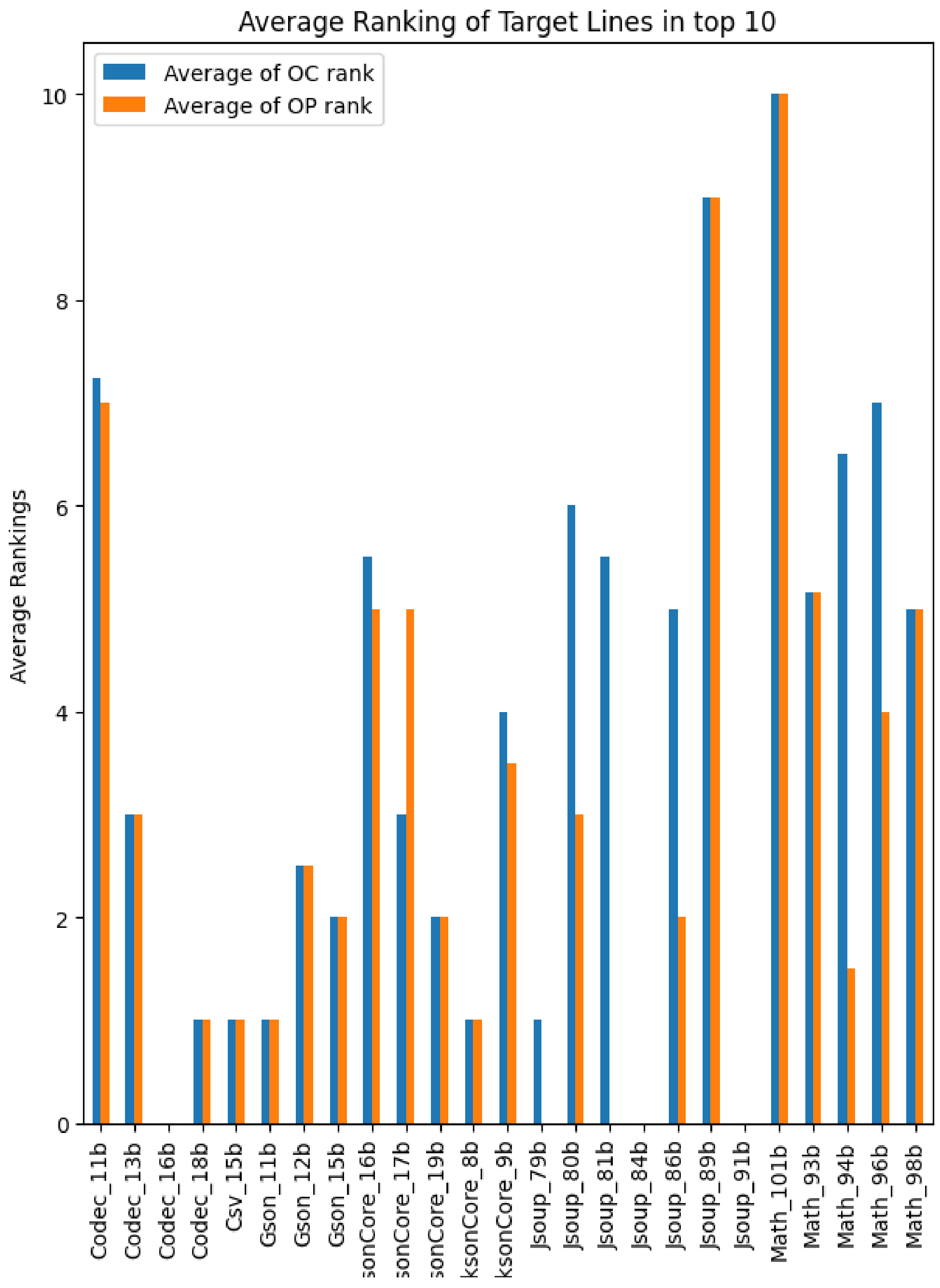


Figure 4.3: Average Ranking of the Target Lines located in the first 10 lines of the SFL reports

Chapter 5

Conclusion

Contents

This Chapter presents some considerations on the work developed during this document and implemented system. Section 5.1 addresses the conclusions drawn from this work. Following, Section 5.2 defines the main contributions retrieved from our work, and finally, Section 5.3 tackle the research possibilities that this document kicked off.

5.1 Conclusion

The software systems are more and more complex, and consequently, the possibility of deep and even shallow failures in the applications has been raised. Although, the failures are not caught during the software development and only appears when the product is already with the end-user, causing lots of losses and possibly human lives injuries. Testing is an essential phase of development. If every flaw is caught, there are no losses to be mitigated. Fuzzing is a field that raised from this necessity of testing intensively to avoid the maximum number of vulnerabilities, and he has shown good results finding them.

The next step after finding the failures is to fix them. Debugging is a tedious and time-consuming task for developers when the software does not show the expected behaviour. The automation of the bug localization improved this process. If the developer can reach the bug location faster in the code or has a path to start with, the time spent on failure and fixing might decrease. Although, this assumes that somehow the failure was already found.

In our work, we address this gap. We tried to connect two processes with demonstrated results separately. Grabbing two state-of-the-art tools of each field, Jazzer from the Fuzzing area and Gzoltar from the SFL field, we connected them through a parser written in Java that translated the tests generated by Jazzer to tests that Gzoltar could use. In addition, we performed a study in order to understand if the connection created may augment the tests creation and bug finding.

Despite the modest results, we could show that this connection is possible under some constraints. We demonstrated through the Defects4J framework that the system finds the bugs defined by the framework and others unknown for that version of the software. Then, the system revealed that with the automatic tests created and translated, the suspicious values and rankings were reached and, in some cases, exceeded them. It is important to mention that were cases where some tests did not reach the intended results due to the limitations of the different tools.

5.2 Main Contributions

This document and research present some contributions to improving the testing phase of software.

Even as an initial study about the topic, we did not find any research or tool that tried to connect Fuzzing and SFL techniques to improve the software testing and, consequently, the improvement of software quality. We believe this pioneering study may open some possibilities in both fields.

Following, although short, we assemble state-of-the-art tools and techniques, gathering together some of the latest tools in Fuzzing and SFL and the well-known and acknowledged ones like that we utilize in the SFL module.

Finally, the small parser we created to connect the Fuzzing and SFL modules, although it cannot be used on an industrial level for obvious reasons, can be used and adapted for new research in these fields.

5.3 Future Work

Despite the humble results, our system has limitations. There are two main issues to be addressed next. One of the steps that took most of the time was creating the Fuzz driver in the first module. For every single test created, we had to create a driver containing the functions we wanted to fuzz to find the specific bug. Further studies may create or use an automatic driver generation, or experiment the auto fuzzing, recently added to Jazzer. In the case of Jazzer, the tool tries to guide the fuzzing campaign targeting the class passed to auto fuzzing mode. The second main issue regards the study's premise, the existence of passing tests. As mentioned in Section 4.3, the passing tests of the developers' suite had to be used in order to Gzoltar retrieve better results. A way to complement the study in a future perspective is to generate correct inputs and not just keep those that crash the software under test.

Furthermore, our system can use Jazzer output failing tests and others if the fuzzer used generates the input and as binary files and if the program under test has a method that allows it to consume this type of files. Extending the parser to read different Fuzzers output would be interesting. This extension would allow the system under test to be scrutinized by different methods and applications, and in the end, pass the information to the SFL module for bug finding.

Different techniques could be added to the SFL module as well. As mentioned in Chapter 2, SFL techniques, and in this specific case, SBFL techniques, are not perfect. Each available technique may perform better than another depending on the environment and variables. Thus, if available in Java or created from scratch, more techniques like D*, for example, could be added to the SFL module and more information regarding the bug locations.

Finally, our system maintains two independent modules, which means that double the computing resources are being consumed. There are similar processes on both sides, such as instrumentation of the program under test, where each one does instrumentation. Thus, instead of keeping both sides independent, the existing run-time overhead could be mitigated if they could share resources. Although, unfolding the components would make the system lose its modules independence, bringing new unpredictable issues.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11): 1780–1792, nov 2009. ISSN 01641212. doi: 10.1016/j.jss.2009.06.035. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121209001319`.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE, nov 2009. ISBN 978-1-4244-5259-0. doi: 10.1109/ASE.2009.25. URL `https://ieeexplore.ieee.org/document/5431781/`.

[3] Rui Abreu, André Riboira, and Franz Wotawa. Constraint-based debugging of spreadsheets. In *15th Ibero-American Conference on Software Engineering, CIbSE 2012*, pages 1–14, 2012.

[4] Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal*, 23(2):297–322, jun 2015. ISSN 15731367. doi: 10.1007/s11219-014-9236-4. URL `http://link.springer.com/10.1007/s11219-014-9236-4`.

[5] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and Reflections. *IEEE Software*, 38(3):79–86, may 2021. ISSN 19374194. doi: 10.1109/MS.2020.3016773.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, New York, NY, USA, oct 2016. ACM. ISBN 9781450341394. doi: 10.1145/2976749.2978428. URL `http://dx.doi.org/10.1145/2976749.2978428https://dl.acm.org/doi/10.1145/2976749.2978428`.

[7] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, may 2013. ISBN 978-1-4673-3076-3. doi: 10.1109/ICSE.2013.6606558. URL `http://ieeexplore.ieee.org/document/6606558/`.

[8] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, pages 378–381, New York, New York, USA, 2012. ACM Press. ISBN 9781450312042. doi: 10.1145/2351676.2351752. URL `http://dl.acm.org/citation.cfm?doid=2351676.2351752`.

[9] Serjei Dechand, Christian Hartlage, Fabian Meumertzheim, Sebastian Pöplau, Mohammed Qasem, Simon Resch, Henrik Schnor, and Khaled Yakdan. Jazzer, 2021. URL `https://github.com/CodeIntelligenceTesting/jazzer`.

[10] Christoph Diehl and Alexander Borgardt. posidron/dharma: Generation-based, context-free grammar fuzzer., 2020. URL `https://github.com/posidron/dharma`.

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 760–770. IEEE, jun 2012. ISBN 978-1-4673-1067-3. doi: 10.1109/ICSE.2012.6227142. URL `http://ieeexplore.ieee.org/document/6227142/`.

[12] Debolina Ghosh and Jagannath Singh. Effective spectrum-based technique for software fault finding. *International Journal of Information Technology (Singapore)*, 12(3):677–682, sep 2020. ISSN 25112112. doi: 10.1007/s41870-019-00347-1. URL `https://link.springer.com/article/10.1007/s41870-019-00347-1`.

[13] Patrice Godefroid. Fuzzing. *Communications of the ACM*, 63(2):70–76, jan 2020. ISSN 0001-0782. doi: 10.1145/3363824. URL `https://dl.acm.org/doi/abs/10.1145/3363824https://dl.acm.org/doi/10.1145/3363824`.

[14] Patrice Godefroid, Michael Y. Levin, and David a. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium (NDSS)*, volume 9. The Internet Society, 2008. ISBN 978-3-642-02651-5. URL `https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/`.

[15] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE. *Communications of the ACM*, 55(3):40–44, mar 2012. ISSN 0001-0782. doi: 10.1145/2093548.2093564. URL `https://dl.acm.org/doi/abs/10.1145/2093548.2093564https://dl.acm.org/doi/10.1145/2093548.2093564`.

[16] Alberto González-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan Gemund. Spectrum-Based Sequential Diagnosis. In *Proceedings of the National Conference on Artificial Intelligence*, volume 1, mar 2011.

[17] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134:340–354, dec 2017. ISSN 01641212. doi: 10.1016/j.jss.2017.09.018. URL `https://linkinghub.elsevier.com/retrieve/pii/S0164121217302066`.

[18] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. *AddressSanitizer: A Fast Address Sanity Checker*. USENIX Association, Boston, 2012. ISBN 978-931971-93-5. URL `https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany`.

[19] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. Empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, 10(3):171–194, 2000. ISSN 09600833. doi: 10.1002/1099-1689(200009)10:.

[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th*

ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 230–243, New York, NY, USA, jul 2021. ACM. ISBN 9781450384599. doi: 10.1145/3460319.3464795. URL https://doi.org/10.1145/3460319.3464795https://dl.acm.org/doi/10.1145/3460319.3464795.

[21] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, pages 273–282, 2005. doi: 10.1145/1101908.1101949.

[22] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 437–440, New York, New York, USA, 2014. ACM Press. ISBN 9781450326452. doi: 10.1145/2610384.2628055. URL http://dl.acm.org/citation.cfm?doid=2610384.2628055.

[23] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, dec 2018. ISSN 2523-3246. doi: 10.1186/s42400-018-0002-y. URL https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y.

[24] Zijie Li, Lanfei Yan, Yuzhen Liu, Zhenyu Zhang, and Bo Jiang. MURE: Making Use of MUtations to REfine Spectrum-Based Fault Localization. In *Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018*, pages 56–63. Institute of Electrical and Electronics Engineers Inc., aug 2018. ISBN 9781538678398. doi: 10.1109/QRS-C.2018.00024.

[25] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, sep 2018. ISSN 0018-9529. doi: 10.1109/TR.2018.2834476. URL https://ieeexplore.ieee.org/document/8371326/.

[26] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, nov 2021. ISSN 0098-5589. doi: 10.1109/TSE.2019.2946563. URL https://ieeexplore.ieee.org/document/8863940/.

[27] Gabriela K. Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley K.G. Assunção. Spectrum-based feature localization: A case study using ArgoUML. In *ACM International Conference Proceeding Series*, volume Part F1716, pages 126–130. Association for Computing Machinery, sep 2021. ISBN 9781450384698. doi: 10.1145/3461001.3473065. URL https://doi.org/10.1145/3461001.3473065.

[28] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, dec 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://dl.acm.org/doi/abs/10.1145/96267.96279https://dl.acm.org/doi/10.1145/96267.96279.

[29] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3), aug 2011. ISSN 1049331X. doi: 10.1145/2000791.2000795. URL https://dl.acm.org/doi/abs/10.1145/2000791.2000795.

[30] Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Variability fault localization. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, volume Part F1716, pages 120–125, New York, NY, USA, sep 2021. ACM. ISBN 9781450384698. doi: 10.1145/3461001.3473058. URL https://doi.org/10.1145/3461001.3473058https://dl.acm.org/doi/10.1145/3461001.3473058.

[31] Spencer Pearson. Evaluation of fault localization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 13-18-Nove, pages 1115–1117, New York, NY, USA, nov 2016. ACM. ISBN 9781450342186. doi: 10.1145/2950290.2983967. URL http://dx.doi.org/10.1145/2950290.2983967https://dl.acm.org/doi/10.1145/2950290.2983967.

[32] Spencer Pearson, Jose Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, may 2017. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.62. URL http://ieeexplore.ieee.org/document/7985698/.

[33] Alexandre Perez, Rui Abreu, and André Riboira. A dynamic code coverage approach to maximize fault localization efficiency. *Journal of Systems and Software*, 90(1):18–28, apr 2014. ISSN 01641212. doi: 10.1016/j.jss.2013.12.036. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121214000090.

[34] Alexandre Perez, Rui Abreu, and Marcelo D'Amorim. Prevalence of Single-Fault Fixes and Its Impact on Fault Localization. In *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*, pages 12–22. IEEE, mar 2017. ISBN 9781509060313. doi: 10.1109/ICST.2017.9. URL http://ieeexplore.ieee.org/document/7927959/.

[35] Henrique Lemos Ribeiro, Higor Amario De Souza, Roberto Paulo Andrioli De Araujo, Marcos Lordello Chaim, and Fabio Kon. Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software. *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pages 404–409, may 2018. doi: 10.1109/ICST.2018.00048.

[36] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. KAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*, pages 167–182, 2017. ISBN 9781931971409. URL https://github.com/RUB-SysSec/kAFL.

[37] André Silva, Matias Martinez, Benjamin Danglot, Davide Ginelli, and Martin Monperrus. FLACOCO: Fault Localization for Java based on Industry-grade Coverage. Technical report, arXiv, nov 2021. URL https://arxiv.org/pdf/2111.12513.pdfhttp://arxiv.org/abs/2111.12513.

[38] Ian Sommerviller. *Software Engineering*. Addison-Wesley, 9th edition, 2011. ISBN 9780137035151.

[39] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software Fault Localization Using DStar (D*). In *2012 IEEE Sixth International Conference on Software Security and*

Reliability, pages 21–30. IEEE, jun 2012. ISBN 978-1-4673-2067-2. doi: 10.1109/SERE.2012.12. URL `https://ieeexplore.ieee.org/document/6258291`.

[40] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, aug 2016. ISSN 0098-5589. doi: 10.1109/TSE.2016.2521368. URL `http://ieeexplore.ieee.org/document/7390282/`.

[41] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. An Empirical Study on the Usage of Fault Localization in Automated Program Repair. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 504–508, New York, New York, USA, sep 2017. IEEE. ISBN 978-1-5386-0992-7. doi: 10.1109/ICSME.2017.37. URL `http://ieeexplore.ieee.org/document/8094451/`.